KATHOLIEKE
UNIVERSITEIT
LEUVEN

DEPARTEMENT TOEGEPASTE ECONOMISCHE WETENSCHAPPEN

RESEARCH REPORT 9949

SCHEDULING PROJECTS WITH LINEAR TIME-DEPENDENT CASH FLOWS TO MAXIMIZE THE NET PRESENT VALUE

by

M. VANHOUCKE
E. DEMEULEMEESTER
W. HERROELEN

D/1999/2376/49

SCHEDULING PROJECTS WITH LINEAR TIME-DEPENDENT CASH FLOWS TO MAXIMIZE THE NET PRESENT VALUE

Mario VANHOUCKE • Erik DEMEULEMEESTER • Willy HERROELEN

November 1999

Operations Management Group
Department of Applied Economics
Katholieke Universiteit Leuven
Naamsestraat 69, B-3000 Leuven (Belgium)
Phones: 32-16-32 69 65, 32-16-32 69 72, 32-16-32 69 70, Fax 32-16-32 67 32
E-mail: <first name>.<name>@econ.kuleuven.ac.be

SCHEDULING PROJECTS WITH LINEAR TIME-DEPENDENT CASH FLOWS TO MAXIMIZE THE NET PRESENT VALUE

Mario VANHOUCKE • Erik DEMEULEMEESTER • Willy HERROELEN

ABSTRACT

In this paper we study the unconstrained project scheduling problem with discounted cash flows where the net cash flows are assumed to be linear dependent on the completion times of the corresponding activities. Each activity of this unconstrained project scheduling problem has a known deterministic net cash flow which is linear and non-increasing in time. Progress payments and cash outflows occur at the completion of activities. The objective is to schedule the activities in order to maximize the net present value (*npv*) subject to the precedence constraints and a fixed deadline. Despite the growing amount of research concerning the financial aspects in project scheduling, little research has been done on the problem with time-dependent cash flows. Nevertheless, this problem gives an incentive to solve more realistic versions of project scheduling problems with financial objectives.

We introduce an extension of an exact recursive algorithm which has been used in solving the max-*npv* problem with time-independent cash flows and which is embedded in an enumeration procedure. The recursive search algorithm schedules the activities as soon as possible and searches for sets of activities to shift towards the deadline in order to increase the net present value. The enumeration procedure enumerates all sets of activities for which such a shift has not been made but could, eventually, have been advantageous. The procedure has been coded in Visual C++ version 4.0 under Windows NT and has been validated on a randomly generated problem set.

Keywords: *Project scheduling; Net present value; Linear time-dependent cash-flows; Optimal search*

1. Introduction

The problem of scheduling activities of a project in order to maximize its net present value has gained increasing attention throughout the literature. In these problems, cash flows occur throughout the life of the project according to a rich variety of possible patterns (positive and/or negative, event-oriented or activity-based). Most of the previous research has focused on the case where the cash flows are independent of the completion of the corresponding activity. For an overview of the literature, we refer to Herroelen et al. (1997) and Vanhoucke et al. (1999). Despite the criticism on this assumption, little work has been done on the case where activity cash flows are dependent on the completion times of the corresponding activities. In a large number of projects, however, the cash flows depend on their occurrence in different ways, resulting in a large number of different time-dependencies.

Dayanand and Padman (1993a, 1993b, 1997) have examined financial project scheduling problems where the negative cash flows are known over the duration of the project but the amount and timing of positive cash flows may be unknown. This problem of simultaneously determining the amount and timing of progress payments in order to maximize the net present value reduces to distributing payments over the duration of the project. In their procedure, milestone events are identified in which the amount paid is proportional to the time elapsed of the project, i.e. the so-called *recovery of expenses.*

Kazaz and Sepil (1996) have presented a MIP formulation with Benders decomposition for a project scheduling problem where the cash flows do not occur at some event realization time, but as progress payments at the end of some time periods. They have defined the so-called *activity profit curve* as a graph showing the net present value change of cash flows associated with an activity with respect to activity finish times. They have shown that the discount rate is the main parameter that affects the overall shape of this curve. Sepil and Ortaç (1997) developed three different heuristic rules to solve the same problem extended with renewable resource constraints.

Etgar et al. (1996) present a simulated annealing solution approach for the case where the event-oriented cash flows are assumed to be a non-increasing step function of time. Shtub and Etgar (1997) developed a branch-and-bound based procedure for the same problem. They have shown that this procedure outperforms the previous simulated annealing procedure with respect to the computational time. Recently, Etgar and Shtub (1999) have extended the work of Elmaghraby and Herroelen (1990) to treat the problem with linear dependent cash flow patterns.

In this paper we present an exact procedure to maximize the net present value in activity-on-the-node project networks with zero-lag finish-start precedence constraints where the activity-based cash flows are linear dependent on the completion times of the corresponding activities (problem $cpm,\delta_n,c_j^{lin} \mid npv$, following the classification scheme of Herroelen et al. (1998) and further denoted as the max-npv^{lin} problem). To the best of our knowledge, the problem has only recently been solved by Etgar and Shtub (1999) for activity-on-the-arc networks. In their paper, the logic of the algorithm is demonstrated by means of an example. They show that the proposed algorithm is a simple extension of the work by Elmaghraby and Herroelen (1990). Unfortunately, the authors do not present computational results.

The organization of the paper is as follows. In section 2 we discuss the max-npv^{lin} problem. Section 3 describes the logic of the exact procedure. Section 4 deals with the introduction of three dominance rules. In section 5 we illustrate the algorithm by means of

an example. Section 6 tests its performance and reports detailed computational results. Section 7 gives an overall conclusion and areas for future research.

2. The max-npv^{lin} problem

The deterministic max-npv^{lin} problem involves the scheduling of project activities within a certain deadline δ_n in order to maximize the net present value (*npv*) of the project in the absence of resource constraints. The project is represented by an activity-on-the-node (AoN) network where the set of nodes, N, represents activities and the set of arcs, A, represents finish-start precedence constraints with a time lag of zero. The activities are numbered from the dummy start activity 1 to the dummy end activity n. The duration of an activity is denoted by d_i ($1 \leq i \leq n$) and the performance of each activity involves a series of cash flow payments and receipts throughout the activity duration. We assume the cash flows of activity i to occur at the end of its execution, which can be easily obtained by compounding the associated cash flows to the end of the activity. We further assume that these cash flows are linear and non-increasing in time.

Since the cash flow of each activity i is assumed to be linear and non-increasing in time, it can be represented by the function $c_i = a_i + b_i f_i$, where the nonnegative integer variable f_i ($1 \leq i \leq n$) denotes the completion time of activity i, $a_i, b_i \in \Re$ and $b_i \leq 0$. Remark that, when $b_i = 0$ for all activities i, the problem reduces to a max-*npv* problem with time-independent cash flows which is the topic of a growing number of publications (Russell (1970), Grinold (1972), Elmaghraby and Herroelen (1990), Herroelen and Gallens (1993)). Recently, Demeulemeester et al. (1996) have introduced a fast recursive search algorithm for solving this time-independent max-*npv* problem. In this case, positive cash flows should be scheduled as early as possible while negative cash flows should be scheduled as late as possible within the precedence constraints. We have extended this idea for the case where $b_i \leq 0$, a case for which no such clear relation exists.

Consider Fig. 1 which shows three net cash flow functions c_i of activity i and their corresponding net present values through time. When $c_i = a_i + b_i f_i$, represents the terminal value of cash flows of activity i at its completion time f_i and α represents the discount rate, its discounted value at the beginning of the project is $G(f_i) = (a_i + b_i f_i)\beta^{f_i}$, with $\beta = \frac{1}{1+\alpha}$.

In both the first and the second case, b_i equals zero which reduces the problem to the original max-*npv* problem without time-dependencies. The third case represents the net cash flow function and the net present value function for the case when $a_i > 0$ and $b_i < 0$. A similar net present value graph can be obtained for the case when $a_i < 0$ and $b_i < 0$.

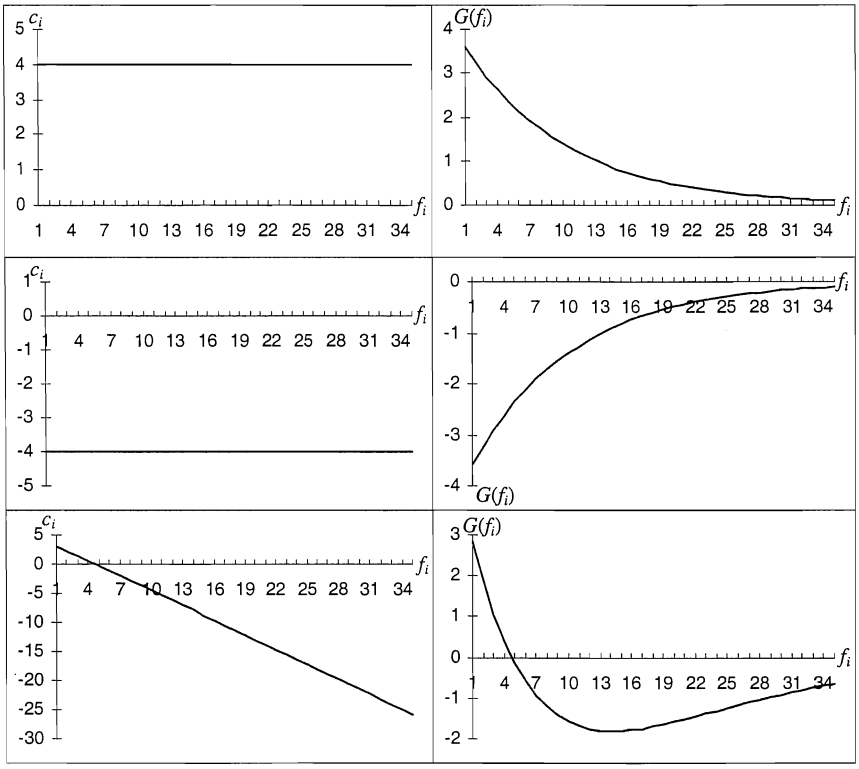

Fig. 1. The net cash flow c_i and its corresponding net present value $G(f_i)$ of activity i where (a) $a_i > 0$ and $b_i = 0$, (b) $a_i < 0$ and $b_i = 0$ and (c) $a_i > 0$ and $b_i < 0$ for $\beta = 0.9$.

A formulation of the max-npv^{lin} problem can be given as follows:

$$\text{Maximize} \sum_{i=2}^{n-1} (a_i + b_i f_i)\beta^{f_i} \qquad [1]$$

Subject to

$$f_i \le f_j - d_j \qquad \forall (i,j) \in A \qquad [2]$$

$$f_n \le \delta_n \qquad [3]$$

$$f_1 = 0 \qquad [4]$$

The objective in Eq. 1 maximizes the net present value of the project. The constraint set in Eq. 2 maintains the finish-start precedence relations among the activities. In order to restrict the project duration, we add a negotiated project deadline δ_n given in Eq. 3. Eq. 4 forces the dummy start activity to end at time zero.

In the next section we discuss the exact solution procedure for the max-npv^{lin} problem as formulated above in Eqs. [1] - [4].

3. The exact algorithm

The exact algorithm for solving the max-npv^{lin} problem consists of two parts: a recursive search procedure which is embedded in an enumeration procedure. The first three steps are part of an extended version of the fast recursive search algorithm for the original max-*npv* problem without time-dependencies of Demeulemeester et al. (1996). These steps are the subject of section 3.1. The fourth step, which is the subject of section 3.2, consists of the enumerative procedure which repeatedly makes use of the recursive search procedure.

3.1 Description of the recursive algorithm

The recursive search algorithm consists of three steps: the construction of the early tree, the construction of the current tree and the generation of the set of delaying trees *DT* by searching the current tree.

Step 1: Constructing the early tree

In the first step, the algorithm starts by building an *early tree* that schedules the activities as early as possible through the use of traditional forward pass critical path calculations. In doing so we make sure that each activity is linked with the dummy start activity 1 along a path in the tree and can only be shifted forward in time (towards the deadline). The dummy end activity n is scheduled at the deadline δ_n and is linked in the early tree with the dummy start activity 1. This arc is needed in order to allow the recursive search which is performed in step 3 to start from the dummy start node.

Step 2: Constructing the current tree

In step 2 we compute the *current tree* by delaying, in reverse order, all activities i which have no successor in the early tree and whose finish time is larger than the point where the net present value is minimized, i.e. $\max(0, \frac{-b_i - a_i \ln\beta}{b_i \ln\beta})$ as shown in theorem 1.

Theorem 1 : The minimum of the function $F(x)=(a+bx)\beta^x$, $b\neq 0$, equals $\frac{-b_i - a_i \ln\beta}{b_i \ln\beta}$.

Proof:

The minimum of the function $F(x)=(a+bx)\beta^x$, $b\neq 0$, equals the solution of the derivative of this function with respect to x. Remark that $\beta^x = e^{x\ln\beta}$.

$$\frac{dF(x)}{dx} = 0$$

$$\Leftrightarrow b\beta^x + (a+bx)e^{x\ln\beta}\ln\beta = 0$$

$$\Leftrightarrow \beta^x(b + (a+bx)\ln\beta) = 0$$

$$\Leftrightarrow x = \frac{-b - a\ln\beta}{b\ln\beta} \text{ or } x = -\infty \text{ (corresponding to a theoretical maximum)}$$

Since the finish time of an activity is a positive integer variable, we set $T=\max(0,\frac{-b_i - a_i \ln\beta}{b_i \ln\beta})$. Notice that, if b=0, the net present value function $F(T)=a\beta^T$ is decreasing in time when $a>0$ or increasing in time when $a<0$ and therefore its minimum point equals ∞ or 0, respectively.

Q.E.D.

The calculations performed in those two steps do not differ very much from the original calculation of the early and current tree for the max-*npv* problem without time-dependencies of Demeulemeester et al. (1996). In the original problem, the current tree is constructed by delaying activities i with a negative cash flow and no successor in the early tree. In this paper, activities i are delayed when $f_i > \max(0,\frac{-b_i - a_i \ln\beta}{b_i \ln\beta})$ as described earlier.

Step 3: Generating the set of delaying trees DT

In step 3, in which the current tree is the subject of a recursive search, a number of modifications are made. As was the case for the original max-*npv* problem, the recursive search procedure identifies sets of activities *SA* that might be shifted forward in time (towards the deadline) in order to increase the net present value of the project. The recursive search for the max-npv^{lin} problem also creates a set of delaying trees *DT* in which sets of activities *SA*, which can eventually be shifted, are stored. In what follows, we explain the logic of the recursive search method. The pseudocode is described at the end of this section.

For each set of activities *SA* the algorithm detects whether or not a shift will lead to an increase of the net present value. If such a shift is advantageous, all activities $i \in SA$ will be shifted within an allowable displacement interval, otherwise the set *SA* is added to the set of delaying trees *DT*. Therefore, the algorithm needs to compute the allowable displacement interval each time a set of activities *SA* is found as follows. Compute $v_{k^*l^*} = \min_{\substack{(k,l)\in A \\ k\in SA \\ l\notin SA}} \{f_l - d_l - f_k\}$ and $w = \max_{k\in SA}\{lf_k - f_k\}$, where lf_i denotes the latest finish time of activity i. Since the algorithm now has information about the displacement interval of the set of activities *SA*, it can detect whether a shift is advantageous or not.

Therefore, we need to examine the combined net present value function of the activities of *SA*. As already mentioned by Etgar and Shtub (1999), a set F of all functions of the form $F(T)=(a+bT)\beta^T$ is *closed for summation*, i.e. if $F(T),G(T)\in F$ then $F(T)+G(T)\in F$. Furthermore, the authors have shown that the set is also *closed for summation transposed in time*, i.e. $F(T)+G(T+\Delta T)\in F$. Consider, for example, the following set of activities $SA = \{1,2\}$ as shown in Fig. 2.

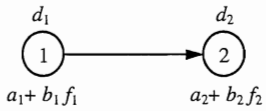

Fig. 2. An example set of activities $SA = \{1,2\}$

In order to examine the net present value function of this set of activities we have to combine both net present value functions into one function as follows:

$G(f_1) = (a_y + b_y f_y)\beta^{f_1}$ with $a_y = a_1 + a_2 \beta^{d_2} + b_2 d_2 \beta^{d_2}$ and $b_y = b_1 + b_2 \beta^{d_2}$ [5]

or

$G(f_2) = (a_y + b_y f_y)\beta^{f_2}$ with $a_y = a_1 + a_2 \beta^{-d_2} - b_2 d_2 \beta^{-d_2}$ and $b_y = b_1 + b_2 \beta^{-d_2}$ [6]

The choice between $G(f_1)$ and $G(f_2)$ is dependent on which activity y we choose as the base of our computations (further denoted as the basic activity y). Remark that [5] is the result of $(a_1 + b_1 f_1)\beta^{f_1} + (a_2 + b_2(f_1 + d_2))\beta^{f_1+d_2}$ while equation [6] can be obtained by simplifying $(a_1 + b_1(f_2 - d_2)\beta^{f_2-d_2} + (a_2 + b_2 f_2)\beta^{f_2}$.

Having created the combined net present value function of the activities $i \in SA$ into one function $G(T)=(a'+b'T)\beta^T$ with basic activity y and having an idea concerning the allowable displacement interval, we can detect whether a shift will be advantageous. Since an activity can only be shifted towards the deadline, we can divide the net present value function into three regions according to the activity finish times as shown in Fig. 3.

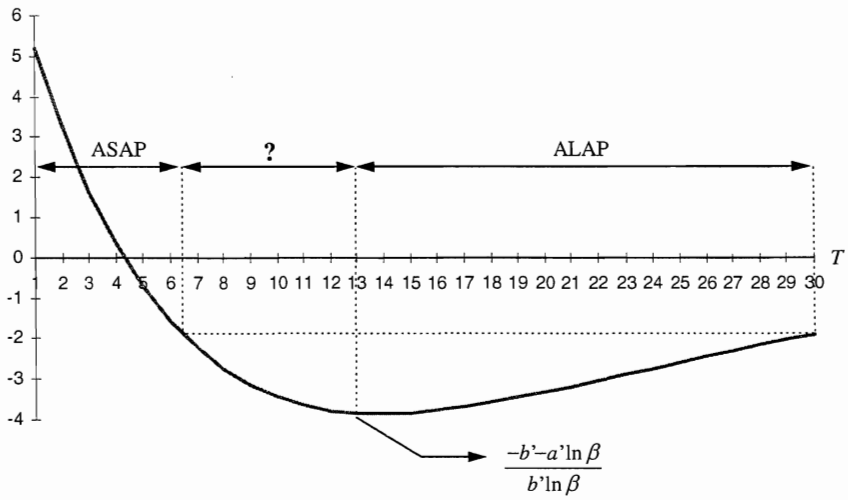

Fig. 3. The net present value of an example SA over time

The leftmost region, denoted by ASAP, corresponds to a region for which the net present value decreases over time. Since in this region a shift of an activity y towards the deadline will never be advantageous it must be scheduled as soon as possible. An activity y

lies in this region when $G(f_y) \geq G(f_y+w)$ (and consequently, since $w \geq v_{k^*l^*}$, $G(f_y) \geq G(f_y+v_{k^*l^*})$). In this case, the algorithm simply continues the recursive search.

The middle region, denoted by the question mark, corresponds to a region for which an activity shift can be advantageous if it can be shifted far enough to increase the net present value. Suppose, for example, that the basic activity of *SA* has a finish time $f_i=8$. It is clear that a shift of the activity towards its latest finish time 30 will lead to an increase in the net present value. If, however (for example due to a successor activity), the activity can only be shifted to time period 20, this will result in a decrease of the net present value. For that reason, we consider two cases. First, if $G(f_y)<G(f_y+v_{k^*l^*})$, a shift will occur. Therefore, the algorithm removes the arc (i,j) which connects a node $i \notin SA$ with a node $j \in SA$ in the current tree *CT* and adds the arc (k^*,l^*) to the current tree *CT*. If l^* belongs to another set of activities *SA'*, i.e. $l^* \in SA' | SA' \in DT$, then the sets *SA* and *SA'* will be merged and the set *DT* will be updated. The completion times of the activities of *SA* are then increased by the allowable displacement $v_{k^*l^*}$ and the algorithm repeats the recursive search. Second, if a shift would be possible but it is prevented by a successor activity, i.e. if $G(f_y)<G(f_y+w)$ (and consequently, $G(f_y) \geq G(f_y+v_{k^*l^*})$ and $v_{k^*l^*}<w$) then the arc (i,j) (with $i \notin SA$ and $j \in SA$) is removed from the current tree *CT* and the set of activities *SA* is added to the set of delaying trees: $DT=DT \cup SA$. Remark that there is no longer a link between the dummy start activity 1 and the activities in the set *DT*. The algorithm repeats the recursive search.

The rightmost region, denoted by ALAP, corresponds to an increasing net present value function and consequently an activity shift will always lead to an increase in the net present value. An activity *y* lies in the rightmost region when $f_y > \frac{-b'-a'\ln\beta}{b'\ln\beta}$ and must therefore be scheduled as late as possible. The algorithm updates the current tree and the set of delaying trees *DT* as described in the first case of the middle region.

The recursive search terminates when no further shift can be found leading to a direct increase of the net present value. The completion times of the activities and of the set of delaying trees *DT* are reported. If the set of delaying trees *DT* is empty then the solution is optimal, else we continue with the enumeration procedure of step 4 which is described in section 3.2. The different steps according to the three different regions are summarized in the pseudocode as shown below.

Pseudocode of the recursive algorithm

When *CA* denotes the set of the activities that are already considered in the recursive search procedure, the pseudocode of the recursive search method can be written as follows:

procedure recursive_search;
$CA=\varnothing$;
Do *RECURSION*$(1) \rightarrow SA', a', b'$;
Report the completion times of the activities, the corresponding net present value, the set of delaying trees *DT* and the current tree *CT*. STOP.

RECURSION(NEWNODE)

Initialize $SA = \{newnode\}$, $a = a_{newnode}$, $b = b_{newnode}$ and $CA = CA \cup \{newnode\}$;

Do $\forall i | i \notin CA$ and i succeeds *newnode* in the current tree CT:

RECURSION$(i) \rightarrow SA'$, a', b';

Compute $v_{k^*l^*} = \min_{\substack{(k,l)\in A \\ k\in SA' \\ l\notin SA'}} \{f_l - d_l - f_k\}\}$ and $w = \max_{k\in SA'} \{lf_k - f_k\}$;

If $G(f_y) < G(f_y + v_{k^*l^*})$ or $f_y > \dfrac{-b' - a' \ln \beta}{b' \ln \beta}$ **then**

Set $CT = CT \setminus (newnode, i)$ and $CT = CT \cup (k^*, l^*)$;

Do $\forall j \in SA'$: set $f_j = f_j + v_{k^*l^*}$;

If $l^* \in SA'' | SA'' \in DT$ **then** merge SA' and SA'' and update DT;

Repeat procedure recursive_search;

Else if $G(f_y) < G(f_y + w)$ **then**

Set $CT = CT \setminus (newnode, i)$ and $DT = DT \cup SA'$;

Repeat procedure recursive_search;

Else Set $SA = SA \cup SA'$ and $a = a + a' * \beta^{d_i} + b' * d_i * \beta^{d_i}$ and $b = b + b' * \beta^{d_i}$;

Do $\forall i | i \notin CA$ and i precedes *newnode* in the current tree CT:

RECURSION$(i) \rightarrow SA'$, a', b';

Set $SA = SA \cup SA'$ and $a = a + a' * \beta^{-d_{newnode}} - b' * d_{newnode} * \beta^{-d_{newnode}}$ and

$b = b + b' * \beta^{-d_{newnode}}$;

Return;

3.2 Description of the enumerative procedure

If upon completion of the recursive search procedure the set of delaying trees DT is not empty, the max-npv^{lin} problem is solved by an enumerative algorithm which is based on the following logic. The binary enumerative procedure selects at each level a set of activities $SA \in DT$ which will either be shifted (first branching node) or not (second twin-node). If in the first branching node the set SA is shifted, an allowable displacement interval $v_{k^*l^*}$ is computed and the finish times of the activities $i \in SA$ are increased by this displacement interval. It is often the case that the net present value of the activities $i \in SA$ decreases due to the shift. Later during the execution of the algorithm, it is quite possible (and hopeful) that these activities will be shifted even further towards the deadline and, as a results, have a greater net present value than was originally the case. In the remainder of this section, we explain the procedure in detail.

Step 4: Enumerating the sets of delaying trees DT

The enumerative procedure can be represented as a binary tree as follows: The algorithm randomly selects at each level p a set of activities SA from DT (which is active) and stores the finish times, the set of delaying trees DT and the current tree CT. The selected set of activities SA consists of a number of activities which are connected to each other by means of a tree. However, since this tree does not have a connection with the dummy start activity 1 along a path through the current tree CT we can either force such a link (first branch) or not (second twin-node branch). Originally, all sets of activities SA are

active. A set SA can be made inactive if it is the subject of a selection in a twin node, as will be explained later on.

In the first node on level p+1, we search for an arc to force such a link. Therefore, an allowable displacement interval $v_{k^*l^*} = \min_{\substack{(k,l)\in A \\ k\in SA \\ l\notin SA}} \{f_l - d_l - f_k\}$ is computed, the finish times of the activities $i \in SA$ are increased by this interval and the arc (k^*,l^*) is added to the current tree CT. If, on the one hand, $l^* \in SA' \in DT$, we merge SA and SA' and update DT. The newly created set of activities $SA''=SA \cup SA'$, which may be the subject of our enumerative procedure later on, does now belong to the set of delaying trees DT and is still not connected with the dummy start activity 1. On the other hand, if l^* is connected along a path with the dummy start activity 1 (and consequently does not belong to a set of activities of DT), the set of activities SA is connected with the dummy start activity 1 through the insertion of arc (k^*,l^*) and can now again be the subject of a shift due to the recursive search procedure. In both cases, the enumerative search procedure will be performed again with the newly established situation.

In the second twin node on level p+1, the algorithm refrains from forcing such a link. Instead, the set of activities SA is now set inactive to prevent that this set wil be selected again deeper in the binary enumeration tree. The algorithm increases the level and the enumerative search procedure will be performed again. Notice that the recursive search procedure will not be performed on the next level since nothing has changed with the current tree CT. Further during the execution of the enumerative procedure, the activities $i \in SA$ can be shifted anyway due to a merge of SA with another set of activities SA'. The algorithm continues its enumerative search at level p+1.

When no sets of activities SA are left unexamined in DT, i.e. $DT=\emptyset$ or all sets $SA \in DT$ are inactive, the net present value will be compared with the currently best one found, i.e. npv_{best} (initially, npv_{best} is initialized to -∞). If the npv is greater than npv_{best}, we save the activity finish times and update npv_{best}. The algorithm backtracks to the previous level in the enumeration, restores the finish times f_i, the current tree CT and the set DT of that level and continues with the twin node (if not yet explored) of the binary enumeration tree. The algorithm stops when we return to the first level of the binary enumeration tree.

The pseudocode for the enumerative procedure for the max-npv^{lin} problem in which the recursive search procedure is embedded can be represented as below. Remark that the enumerative procedure is represented recursively such that the level of the enumeration is stored automatically.

procedure max-npv^{lin}(node);
  If (node equals 1) **then Do** recursive_search $\rightarrow$ DT, npv;
  If $\exists SA \in DT \backslash SA$ is active **then**
    Store $(f_i \mid i \in N, CT, DT)$;
    {first branch}
    $DT=DT \backslash SA$;
    Compute $v_{k^*l^*} = \min_{\substack{(k,l)\in A \\ k\in SA \\ l\notin SA}} \{f_l - d_l - f_k\}$ and set $CT=CT \cup \{k^*,l^*\}$;
    If $l^* \in SA' \mid SA' \in DT$ **then** update DT (SA and SA' are merged into $SA''=SA \cup SA'$);
    Do procedure max-npv^{lin}(1);
    Restore $(f_i \mid i \in N, CT, DT)$;
    {second branch}

Make $SA \in DT$ inactive;
Do procedure max-$npv^{lin}(2)$;
Restore $(f_i \mid i \in N, CT, DT)$;
Else If $npv > npv_{best}$ **then** Save $(npv_{best}=npv, f_i \mid i \in N)$;
Return;

4 Dominance rules for the enumerative procedure

Notice that, due to the logic of the enumerative procedure, the enumeration results in an unbalanced tree. In doing so the procedure creates a large amount of nodes leading to the same optimal solution. In order to reduce the number of nodes in the enumerative procedure, three dominance rules are developed.

4.1 Right-shift dominance rule

Dominance Rule 1 (Right-shift) : $\forall SA \in DT | G(f_y) < G(f_y + v_{k'l'}) : f_i = f_i + v_{k'l'} | i \in SA$ and update DT.

During the recursive search, a number of sets of activities SA are added to the set of delaying trees DT. At their creation, these sets SA represent a number of activities with basic activity y for which $G(f_y) < G(f_y + w)$ and $G(f_y) \geq G(f_y + v_{k*l*})$ with $v_{k*l*} < w$. Later, during the performance of the recursive search, it is possible that $G(f_y) < G(f_y + v_{k'l'})$ and consequently, the activities $i \in SA$ can be shifted anyway. This occurs when certain activities which are successors of the activities $i \in SA$ are shifted after the creation of SA and therefore, the allowable displacement interval has become larger than was originally the case, i.e. $v_{k'l'} > v_{k*l*}$. After the completion of the recursive search procedure at each node, the occurrence of such sets SA will be detected. If this is the case, the activities $i \in SA$ will be shifted, i.e. $f_i = f_i + v_{k'l'} | i \in SA$ and the set of delaying trees DT will be updated as follows: If $l' \in SA' | SA' \in DT$ then DT will be updated by merging SA and SA', otherwise the set SA will be removed from DT. Of course, the recursive search will be repeated before starting the enumerative procedure.

4.2 Cycle detection

Dominance Rule 2 (Cycle-detection) : If a cycle $C=<SA_s, SA_{s+1}, SA_{s+2}, ..., SA_{s+t}>$, $C \subset DT$, exists then merge all $SA \in C$ and update DT.

In what follows, we define a *path* as $<SA_s, SA_{s+1}, SA_{s+2}, ..., SA_{s+t}>$ for which $\exists k \in SA_{s+i}$ and $\exists l \in SA_{s+i+1} | f_l - d_l - f_k = 0$. A path is called a *cycle* if $s=t$. If such cycles exist among the sets of activities $SA \in DT$, these sets can be merged which results in a reduction of the number of elements in DT and consequently, the number of nodes in the enumeration tree. Suppose, for example, that the two sets of $DT=\{SA, SA'\}$ can be merged since $<SA, SA'>$ is a cycle, i.e. $\exists i \in SA$ and $\exists j \in SA' | f_j - d_j - f_i = 0$ and $\exists k \in SA'$ and $\exists l \in SA | f_l - d_l - f_k = 0$, then the updated set of delaying trees $DT=\{SA''=SA \cup SA'\}$ contains only one set of activities SA''.

4.3 Twin-node dominance

Dominance Rule 3 (Twin-node dominance) : If $\exists SA \in DT | \Delta G_1 > 0$ and the insertion of an arc (k^*,l^*) in the first branch at level p has resulted in $\Delta G_1 > \Delta G_2$ then it is not necessary to consider the twin node at level p.

As already mentioned, the enumerative procedure can be represented as a binary tree. At each level, the procedure selects a set of activities SA which will be either shifted towards the deadline (first branch node) or not (second twin-node). If one can show that the shift in the first branch node (by means of the insertion of an arc (k^*,l^*)) has led to the exact solution of the subree rooted at the parent node, then the second twin-node branch must not be considered.

Therefore, let cn be the current node, i.e. the first branch node at level p in the enumeration tree and ln the best leaf node (i.e. the leaf node with the highest npv) from the subtree rooted at this first branching node cn, as shown in Fig. 4. Further, let $\bar{S}_{SA}$ denote the set of direct and indirect successors of the activities $i \in SA$ and $G(SA)_x$ denote the net present value of the activities $i \in SA$ at node x in the enumeration tree (in the following, x will be either cn or ln).

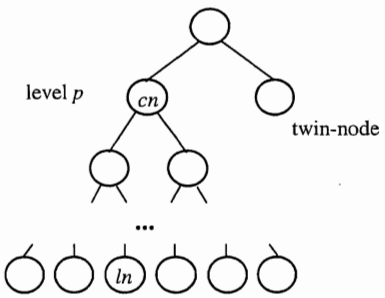

Fig. 4. Illustration of the twin-node dominace rule

If the insertion of an arc (k^*,l^*) in the first branch node at level p has led to an increase in the net present value of the activities $i \in SA$ then $G(SA)_{cn} < G(SA)_{ln}$, i.e. $\Delta G_1 = G(SA)_{ln} - G(SA)_{cn} > 0$. Of course, other activities can be forced to shift as well, due to the shift of the set SA. If Sd_{SA} denotes the set of successor activities of $i \in SA$ for which the net present value has decreased, i.e. $Sd_{SA} = \{i | i \in \bar{S}_{SA} \text{ and } G(i)_{cn} > G(i)_{ln}\}$, the total decrease in the net present value of this set Sd_{SA} is represented by $\Delta G_2 = |G(Sd_{SA})_{ln} - G(Sd_{SA})_{cn}|$.

If the increase in the npv of the set SA, i.e. $\Delta G_1 | \Delta G_1 > 0$, is greater than the resulting decrease ΔG_2 then the shift of SA was advantageous and therefore, the twin-node branch can be dominated.

Proof:

The proof consists of two parts. First, it is shown that the two mentioned conditions are necessary in order to dominate the twin-node branch and secondly, we show that this dominance rule will never prevent the procedure to find the exact solution.

1. Necessity of the conditions $\Delta G_1>0$ and $\Delta G_1>\Delta G_2$.

• Suppose the insertion of an arc (k^*,l^*) in the first branch has not led to an improvement of the net present value of SA, i.e. $\Delta G_1<0$. It must be clear that, in this case, it is necessary to consider the twin-node since it is possible that the net present value is higher without the shift.

• Suppose now that $\Delta G_1<\Delta G_2$ (and $\Delta G_1>0$). Again, it is easy to show that, in certain cases, fathoming the twin-node branch will prevent the procedure to find the exact solution. Consider the following example of Fig. 5 with a deadline $\delta_5=26$. After the performance of the recursive search procedure at the parent node of the enumerative tree, the procedure reports the npv=1.12, the finish times $f_1=0$, $f_2=14$, $f_3=7$, $f_4=15$ and $f_5=26$ and the set of delaying trees $DT=\{(3)\}$. The first branching node at the next level shifts this set of activities $SA=\{3\}$ towards $f_3=14$ and adds the arc (3,4) to the current tree CT. Since now the combined net present value of the activities 3 and 4 with basic activity 3 is $G(f_3)=11-1.25f_3$ the recursive search procedure will shift these activities as far as possible towards the deadline with finish times $f_3=25$ and $f_4=26$. Since $DT=\varnothing$ the solution will be saved as the currently best found solution, i.e. $npv_{best}=0.83$. Notice that the net present value $G(f_3)=(2-1.25f_3)\beta^{f_3}$ of the set $SA=\{3\}$ has increased from -3.22 ($f_3=7$) to -2.09 ($f_3=25$) and consequently $\Delta G_1=-2.09-(-3.22)=1.13$. However, $npv_{best}=0.83$ is lower than the new found net present value npv=1.12 at the parent node. Therefore, it is necessary to consider the second twin-node branch, in which the set $SA=\{3\}$ will not be shifted, which leads to the exact solution, i.e. $npv_{best}=1.12$. Of course, this solution was also found at the parent node. Remark that $\Delta G_2=|G(4|f_4=26)-G(4|f_4=15)|=|0.65-2.06|=1.41$, which is greater than ΔG_1. Therefore, the second condition $\Delta G_1>\Delta G_2$ is necessary in order to apply the twin-node dominance rule.

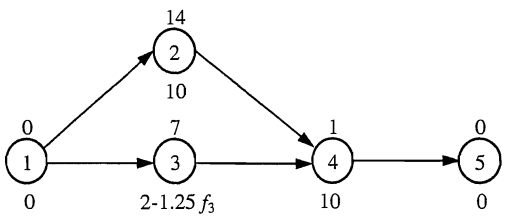

Fig. 5. The net present value of an example SA over time

2. In what follows, we show that applying the twin-node dominance rule will never fail to find the exact solution. It is sufficient to show that the solution that will be found starting from the twin-node can not result in a higher net present value for the project than the one already found by exploring the first branching node. We can distinguish two cases:

***Case 1**: The set of delaying trees DT contains only one set of activities SA, i.e. DT={SA}*

Since $\Delta G_1>0$, the insertion of an arc (k^*,l^*) in the first branch has led to an increase in the net present value of the activities $i\in SA$. Suppose now that, due to the shift, $DT=\varnothing$ (the case where $DT\neq\varnothing$, i.e. at least one other set of activities SA' is created by the recursive search procedure after the shift, is treated in the second case). Moreover, since $\Delta G_1>\Delta G_2$, this increase was stronger than the resulting

decrease in the net present value of other activities. Therefore, the total net present value of the project has increased and the twin-node needs not to be considered.

Case 2: The set of delaying trees DT contains, among SA, also other sets of activities SA', i.e. DT={SA,SA',SA",...}

Since now *DT* contains more than one set of activities *SA*, the resulting decrease in the net present value ΔG_2 is not only a result of the shift of the activities $i \in SA$, but also a result of a shift of other sets of activities in the underlying nodes of the enumeration tree. Therefore, $\Delta G_2=\Delta G_{2,1}+\Delta G_{2,2}$, with $\Delta G_{2,1}$ the decrease in the *npv* due to the shift of *SA* and $\Delta G_{2,2}$ the decrease in the *npv* due to the shift of other sets of activities (notice that $\Delta G_{2,2}=0$ in the first case). If $\Delta G_1>\Delta G_2$, then certainly $\Delta G_1>\Delta G_{2,1}$ and therefore, the increase ΔG_1 is greater than its resulting decrease and consequently, the shift was advantageous.

Q.E.D.

Since this dominance rule results in a considerable decrease of the number of nodes in the enumerative procedure, the algorithm calculates the net present value $G(SA)_{cn}$ of the selected set of activities *SA* at each level *p*. Moreover, the algorithm searches for each node its best leaf node *ln* and stores its corresponding net present value $G(SA)_{ln}$. The same data is stored for the activities $i \in Sd_{SA}$. In doing so, the algorithm tries to dominate the twin-node branch in the enumeration tree. The algorithm still randomly selects the sets $SA \in DT$.

4.4 Remarks on the enumerative procedure *{not in submitted paper}*

In order to gain additional insight in the problem, a few remarks on the logic of the enumerative procedure are worth considering.

The binary enumerative procedure selects at each level a set of activities $SA \in DT$ which will either be shifted (first branching node) or not (second twin-node). If in the first branching node the set *SA* is shifted, an allowable displacement interval $v_{k^*l^*}$ is computed and the finish times of the activities $i \in SA$ are increased by this displacement interval. It is often the case that the net present value of the activities $i \in SA$ decreases due to the shift. Later during the execution of the algorithm, it is quite possible (and hopeful) that these activities will be shifted even further towards the deadline and, as a results, have a greater net present value than was originally the case. From this point of view, one can consider the case in which a set of activities *SA* will be shifted with *u* instead of $v_{k^*l^*}$, with *u* a displacement interval for which $G(f_y) \leq G(f_y+u)$ and $G(f_y)>G(f_y+u-1)$. In doing so, one claims that the net present value of the project can increase only if the net present value of the set of activities *SA* that will be shifted increases. In the example of Fig. 6, the basic activity *y* of *SA* finishes at time 8. The procedure as described in the earlier sections finds the allowable displacement interval $v_{k^*l^*}=10$ and shifts the activities $i \in SA$ with that interval. As a result, the basic activity *y* finishes at time event 18 which leads to a decrease in the net present value. Later on, these activities will (hopefully) be shifted further resulting in an increase of the net present value. If the displacement interval amounts to $u=16$, the net present value of the activities $i \in SA$ is equal to the original *npv*.

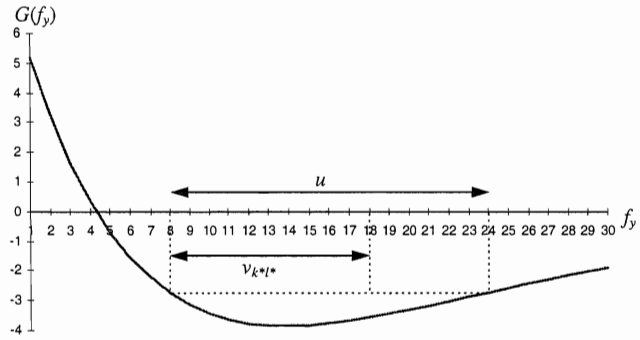

Fig. 6. The net present value of an example *SA* with basic activity *y*

In order to demonstrate that this logic can fail to find the optimal solution, consider the following example of Fig. 7. The network has four activities (and two dummy activities) and a project deadline δ_n=50. The recursive search procedure at the parent node of the enumerative procedure terminates with activity finish times f_1=0, f_2=10, f_3=5, f_4=20, f_5=10 and f_6=20, a *npv*=15.26 and a set of delaying trees DT={(3,5)}. The set of activities $SA \in DT$ with basic activity y=3 has a combined net present value function $G(f_y)$=(-1.14-2.77f_3) β^{f_3} and finishes at time instant 5. The displacement intervals $v_{3,4}$=5 and u=10. On the one hand, the original exact procedure shift the activities 3 and 5 with the allowable displacement interval $v_{3,4}$=5 which leads, in a recursive search on the next level, to a shift of activity 5 to finish time f_5=50. This results in an exact solution with finish times f_1=0, f_2=10, f_3=10, f_4=20, f_5=50 and f_6=50 and a net present value of 16.53. If, on the other hand, the algorithm uses the new displacement interval u=10, then the activities 3 and 5 are shifted towards finish times f_3=15 and f_5=20 and consequently, also activity 4 is forced to shift towards the deadline with finish time f_4=25. The recursive search algorithm on the next level shifts these activities even further to reach a mamixum net present value of 16.36 with finish times f_1=0, f_2=10, f_3=40, f_4=50, f_5=50 and f_6=50. Clearly, this logic fails to find the exact solution.

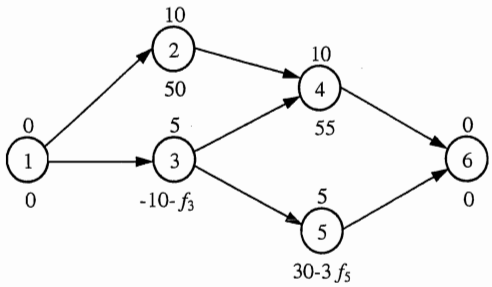

Fig. 7. An example network

5. A numerical example

Consider the AON project network with 11 non-dummy activities given in Fig. 8 and its corresponding net cash flow functions as shown in Table I. The duration of each activity i is denoted above the node and the project deadline δ_n amounts to 30. In order to clarify the logic of the exact algorithm, both the steps of the recursive search procedure and the enumeration are illustrated below (β=0.9).

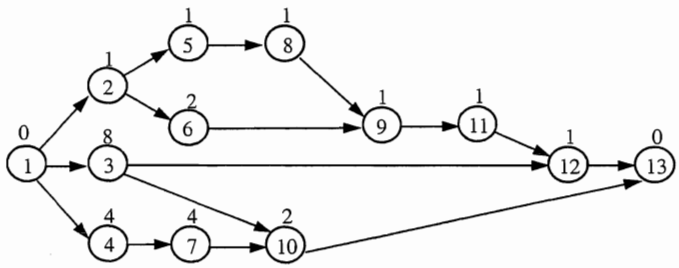

Fig. 8. An example project network

Table I. The linear time-dependent cash flows

i	*net cash flow* c_i		$\max(0, \frac{-b'-a'\ln\beta}{b'\ln\beta})$	lf_i
1	0		0	24
2	-3-0.5f_2	?	4	25
3	5	ASAP	∞	28
4	-5-f_4	?	5	24
5	-5-f_5	?	5	26
6	-0.5	ALAP	0	27
7	-0.5	ALAP	0	28
8	1	ASAP	∞	27
9	1	ASAP	∞	28
10	1	ASAP	∞	30
11	-1-0.25f_{11}	ALAP	6	29
12	4	ASAP	∞	30
13	0		0	30

5.1 The recursive search procedure

The algorithm starts by computing the completion times and the set *DT* by means of the recursive search procedure. The early tree, as shown in Fig. 9, schedules all activities as soon as possible within the precedence constraints. The current tree *CT* is shown in Fig. 10, in which activity 7 is shifted forward in time (with $v_{7,10}$=0).

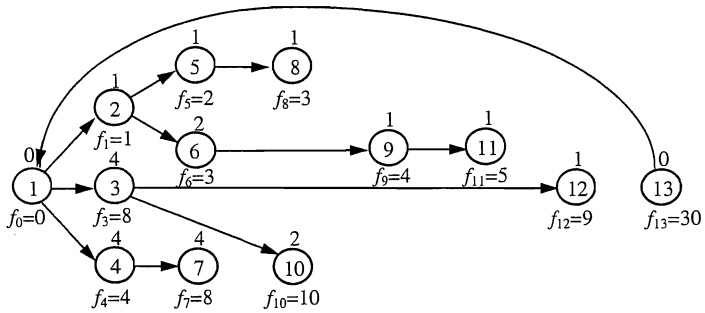

Fig. 9. The early tree of our example project network

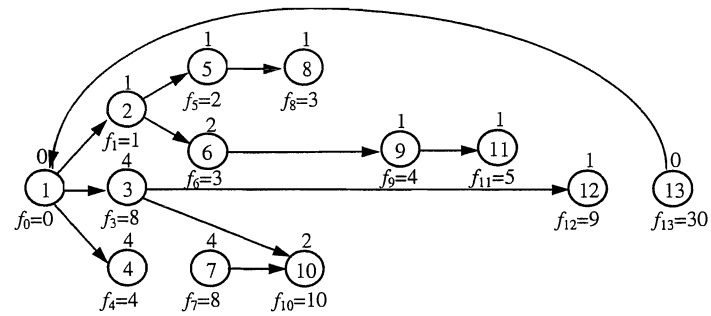

Fig. 10. The current tree of our example project network

The different steps of the recursive_search procedure at the first level of the enumeration are performed as follows. Remark that the current tree *CT* is modified, only when $G(f_y)<G(f_y+v)$, $f_y>\dfrac{-b'-a'\ln\beta}{b'\ln\beta}$ or $G(f_y)<G(f_y+w)$ resulting in a shift of a set of activities *SA* and/or the update of the set *DT*. For the ease of representation, we only mention the steps in which such modifications are made without showing the details of the recursive search through the current tree *CT*. For a detailed description of the recursive_search, we refer the reader to the appendix.

In the following, $G(f_y)=(a'+b'f_y)\beta^{f_y}$ represents the combined net present value function of the set of activities *SA* with basic activity *y*.

- $SA'=\{4\}$, $v=0$, $w=20$ and $y=4$ with $a'=-5.00$ and $b'=-1.00$
  $G(f_4)<G(f_4+w)$, $CT=CT\setminus\{1,4\}$
  Update $DT=\{(4)\}$
- $SA'=\{11\}$, $v=3$, $w=24$ and $y=11$ with $a'=-1.00$ and $b'=-0.25$
  $G(f_{11})<G(f_{11}+v)$, $CT=CT\setminus\{9,11\}$
  Update $CT=CT\cup\{11,12\}$ and $f_{11}=8$
- $SA'=\{5,8\}$, $v=0$, $w=24$ and $y=5$ with $a'=-4.10$ and $b'=-1.00$
  $G(f_5)<G(f_5+w)$, $CT=CT\setminus\{2,5\}$
  Update $DT=\{(4),(5,8)\}$
- $SA'=\{2,6,9\}$, $v=0$, $w=24$ and $y=2$ with $a'=-2.68$ and $b'=-0.50$

$G(f_2)<G(f_2+w)$, $CT=CT\setminus\{1,2\}$
Update $DT=\{(4),(5,8),(2,6,9)\}$

No further modifications can be found and the recursive search procedure terminates. The finish times of the activities as denoted below the node, the current tree *CT* and the set of delaying trees *DT* are shown in Fig. 11. Since $DT\neq\emptyset$ the binary search will be started. Notice that, each time the recursive search procedure will be called, we do not mention any detail. In section 5.2, we show the binary tree without the use of the aforementioned dominance rules. Section 5.3 illustrates the use of these dominance rules.

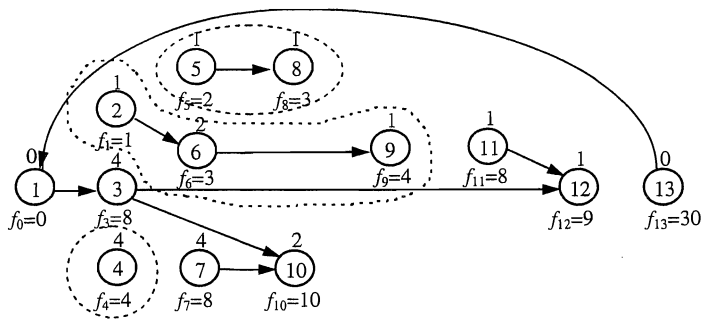

Fig. 11. The current tree *CT* and the set *DT* of the example project after one recursive search

5.2 Enumeration tree without dominance rules

1 recursive_search → $DT=\{(4),(5,8),(2,6,9)\}$
→ $f=\{0, 1, 8, 4, 2, 3, 8, 3, 4, 10, 8, 9, 30\}$
Randomly select $SA=\{4\}$
1.1 Compute $v_{4,7}=0$, $CT=CT\cup\{4,7\}$
2 recursive_search → $DT=\{(5,8),(2,6,9)\}$,
→ $f=\{0, 1, 8, 24, 2, 3, 28, 3, 4, 30, 8, 9, 30\}$
Randomly select $SA=\{5,8\}$
2.1 Compute $v_{8,9}=0$, $CT=CT\cup\{8,9\}$
3 recursive_search → $DT=\{(2,5,6,8,9)\}$,
→ $f=\{0, 1, 8, 24, 2, 3, 28, 3, 4, 30, 8, 9, 30\}$
Select $SA=\{2,5,6,8,9\}$
3.1 Compute $v_{9,11}=3$, $CT=CT\cup\{9,11\}$ and increase the finish times of the activities in *SA* by 3 time units
4 recursive_search → $DT=\emptyset$,
→ $f=\{0, 25, 8, 24, 26, 27, 28, 27, 28, 30, 29, 30, 30\}$
npv=**-3.40**>npv_{best} : Save (npv_{best}=-3.40, $f_i \mid i\in N$)
3.3 twin node
4 no recursive_search → $DT=\emptyset$
npv=-7.69<npv_{best}
2.2 twin node

3 Select $SA=\{2,6,9\}$

3.1 Compute $v_{2,5}=0$, $CT=CT\cup\{2,5\}$

4 recursive_search $\rightarrow DT=\{(2,5,6,8,9)\}$,

$\rightarrow f=\{0, 1, 8, 24, 2, 3, 28, 3, 4, 30, 8, 9, 30\}$

Select $SA=\{2,5,6,8,9\}$

4.1 Compute $v_{9,11}=3$, $CT=CT\cup\{9,11\}$ and increase the finish times of the activities in SA by 3 time units

5 recursive search $\rightarrow DT=\varnothing$,

$\rightarrow f=\{0, 25, 8, 24, 26, 27, 28, 27, 28, 30, 29, 30, 30\}$

$npv=-3.40<npv_{best}$

4.2 twin node

5 no recursive_search $\rightarrow DT=\varnothing$

$npv=-7.69<npv_{best}$

3.2 twin node

4 no recursive_search $\rightarrow DT=\varnothing$

$npv=-7.69<npv_{best}$

1.2 twin node

2. Randomly select $SA=\{5,8\}$

2.1 Compute $v_{8,9}=0$, $CT=CT\cup\{8,9\}$

3 recursive search $\rightarrow DT=\{(2,5,6,8,9)\}$,

$\rightarrow f=\{0, 1, 8, 4, 2, 3, 8, 3, 4, 10, 8, 9, 30\}$

Select $SA=\{2,5,6,8,9\}$

3.1 Compute $v_{9,11}=3$, $CT=CT\cup\{9,11\}$

4 recursive search $\rightarrow DT=\varnothing$,

$\rightarrow f=\{0, 25, 8, 4, 26, 27, 8, 27, 28, 10, 29, 30, 30\}$

$npv=-6.87<npv_{best}$

3.2 twin node

4 no recursive_search $\rightarrow DT=\varnothing$

$npv=-11.16<npv_{best}$

2.2 twin node

3. Select $SA=\{2,6,9\}$

3.1 Compute $v_{2,5}=0$, $CT=CT\cup\{2,5\}$

4 recursive search $\rightarrow DT=\{(2,5,6,8,9)\}$,

$\rightarrow f=\{0, 1, 8, 4, 2, 3, 8, 3, 4, 10, 8, 9, 30\}$

Select $SA=\{2,5,6,8,9\}$

4.1 Compute $v_{9,11}=3$, $CT=CT\cup\{9,11\}$

5 recursive search $\rightarrow DT=\varnothing$,

$\rightarrow f=\{0, 25, 8, 4, 26, 27, 8, 27, 28, 10, 29, 30, 30\}$

$npv=-6.87<npv_{best}$

4.2 twin node

5 no recursive_search $\rightarrow DT=\varnothing$

$npv=-11.16<npv_{best}$

3.2 twin node

4 no recursive_search $\rightarrow DT=\varnothing$

$npv=-11.16<npv_{best}$

The optimal net present value, as shown in bold in the enumeration example, amounts to -3.40 with the corresponding activity completion times: f_1=0, f_2=25, f_3=8, f_4=24, f_5=26, f_6=27, f_7=28, f_8=27, f_9=28, f_{10}=30, f_{11}=29, f_{12}=30 and f_{13}=30. The search tree is shown in Fig. 12.

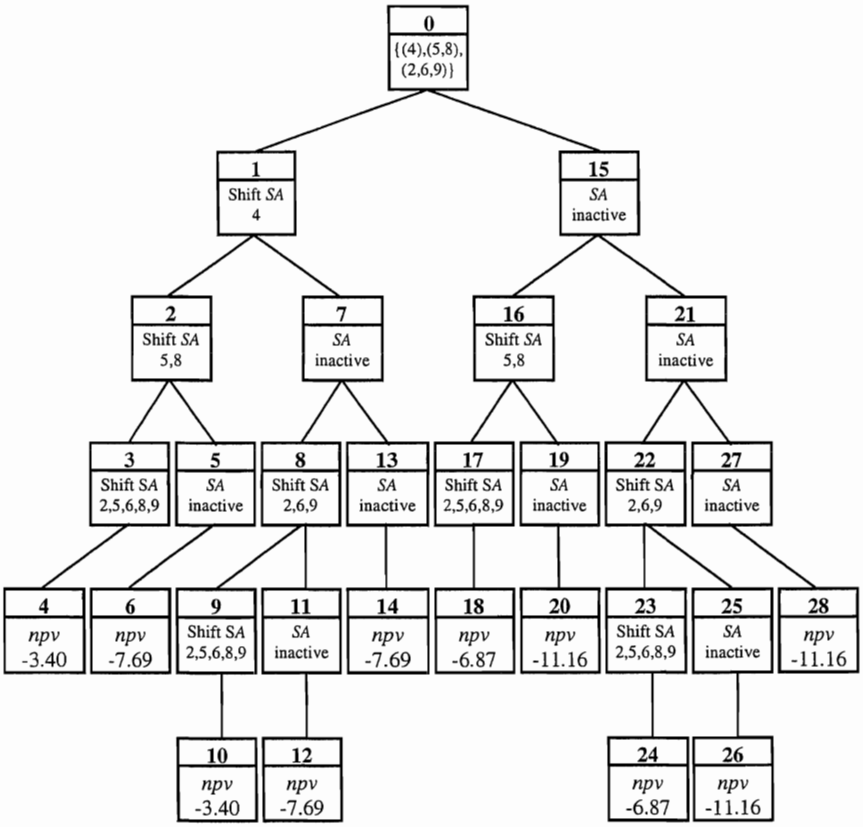

Fig. 12. The example enumeration tree

5.3 The introduction of dominance rules

In what follows, we show that the use of the cycle-detection dominance rule and the twin-node dominance rule results in a considerable decrease in the number of nodes.

Remark that the set of delaying trees *DT* consists of three sets of activities *SA* at node 0 of the enumertion tree, i.e. *DT*={(4),(5,8),(2,6,9)}. Since there exists a cycle between *SA*={5,8} and *SA'*={2,6,9} these sets can be merged by the cycle-detection dominance rule. In fact, f_9-d_9-f_8=0 and f_5-d_5-f_2=0 and therefore a shift of *SA* would result in a shift of *SA'* and vice versa. The application of this dominance rule results in an updated set of delaying trees *DT*={(4),(2,5,6,8,9)} at node 0 as shown in the root node of Fig. 13. This leads to a considerable decrease of the number of nodes in the enumeration tree.

The application of the twin-node dominance rule decreases even further the number of nodes in the enumeration tree. At node 1, *SA*={4} is selected with $v_{4,7}$=0 and the arc (4,7) is added in the current tree. The net present value of this set *SA* at this node

amounts to $G(SA)_{node1}=(-5-4)\beta^4=-5.90$ while the net present value of $SA=\{4\}$ at node 3 (the best leaf node from the subtree rooted at node 1) amounts to $G(SA)_{node3}=(-5-24)\beta^{24}=-2.31$ and consequently, $\Delta G_1=-2.31-(-5.90)=3.59>0$. Notice also that $Sd_{SA=\{4\}}=\{10\}$ (since the net present value of successor activity 7 has increased) with $\Delta G_2=|G(10|f_{10}=30)-G(10|f_{10}=10)|=|0.04-0.34|=0.3$, and therefore, $\Delta G_1>\Delta G_2$. Taken these observations into consideration, node 5 needs not to be considered due to the twin-node dominance rule. A same logic can be followed at level 2 for node 2: applying the twin-node dominance rule results in the domination of node 4. The remaining enumeration tree, as given in Fig. 13, consists only of 4 nodes instead of 26.

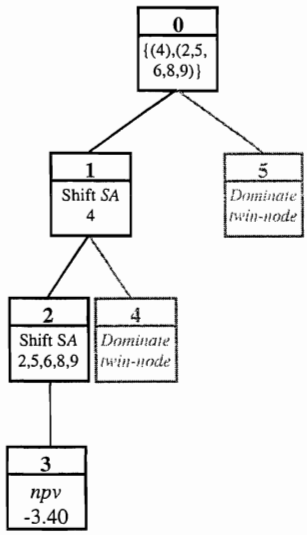

Fig. 13. The remaining enumeration tree after the application of the dominance rules

6. Computational experience

The enumerative procedure for the max-npv^{lin} problem has been coded in Visual C++ Version 4.0 under Windows NT 4.0 on a Dell personal computer (Pentium III 550 MHz processor). In order to validate our procedure, we downloaded the well-known benchmark instances generated by *ProGen* (Kolisch et al., 1995) from the project scheduling library (PSPLIB) developed by Kolisch and Sprecher (1996) at *http://www.bwl.uni-kiel.de/Prod/psplib/index.html*. These instances in activity-on-the-node format use four settings for the number of activities and three settings for the coefficient of network complexity (*CNC*, defined as the number of precedence relations divided by the number of activities) as described in Table II. The sets with 30, 60 and 90 activities contain 480 instances whereas the set with 120 activities contains 600 instances. We deleted the resource requirements and availabilities and provided the problems with a project deadline δ_n and cash flows as follows. In order to measure the impact of the linear dependent and non-increasing cash flow functions $G(f_i)=(a_i+b_i f_i)\beta^{f_i}$ we use different settings for both a_i and b_i as described in Table II. Notice that, when $b_i=0$ for each activity i, the problem reduces to the max-*npv* problem with time-independent cash flows which

can be solved by the recursive search procedure of Demeulemeester et al. (1996). The project deadline was set equal to the critical path length plus a number, as given in Table II.

Table II. Parameter settings used to generate the test instances

Number of activities N	30, 60, 90 or 120
CNC (Kolisch et al, 1995)	1.5, 1.8 or 2.1
Percentage of $a_i<0$	25, 50 or 75
Percentage of $b_i=0$	0, 25, 50, 75 or 100
Project deadline	critical path length + 5, 10 or 15

Table III represents the average CPU-time and its standard deviation in *milliseconds*. Even instances with 120 activities can be solved in a very small amount of time. Notice also the relatively small standard deviations, reflecting the rather robust behaviour of the procedure over the different problem instances.

Table III. Impact of the number of activities

# activities	# problems	Average CPU-time	Standard deviation
30	21,600	0.59	0.09
60	21,600	2.62	2.03
90	21,600	7.25	9.71
120	27,000	11.95	10.64

The results of the impact of the *CNC* are shown in Fig. 14: the higher the *CNC*, the more difficult the problem. However, these results must be interpreted with care, as already pointed out by De Reyck and Herroelen (1996).

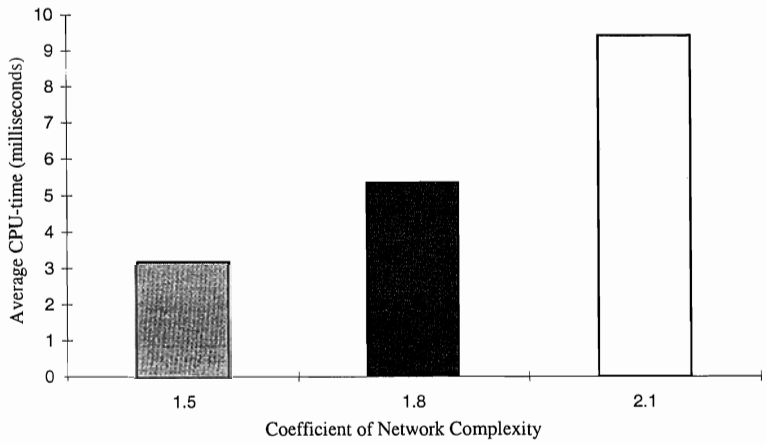

Fig. 14. The impact of the *CNC* on the average CPU-time

Fig. 15 displays the impact of the percentage of $a_i<0$ and $b_i=0$ on the average CPU-time. The five different settings for $b_i=0$ are shown on the X-axis while for each setting of b_i the three different settings for $a_i<0$ are displayed by the three bars. First, the graph

shows that the higher the percentage of a_i<0, the easier the problem. It must be clear that, when a_i≥0, the probability that the finish time f_i lies in the ?-zone (cfr. Fig. 3) is higher than when a_i<0. When a_i<0, it is likely that the finish time f_i lies in the ALAP zone and consequently, less sets of activities *SA* will be added to the set of delaying trees *DT*. Secondly, the graph displays that the higher the percentage of b_i=0, the more difficult the problem, except when all b_i=0 where the problem reduces to the original max-*npv* problem without time-dependencies.

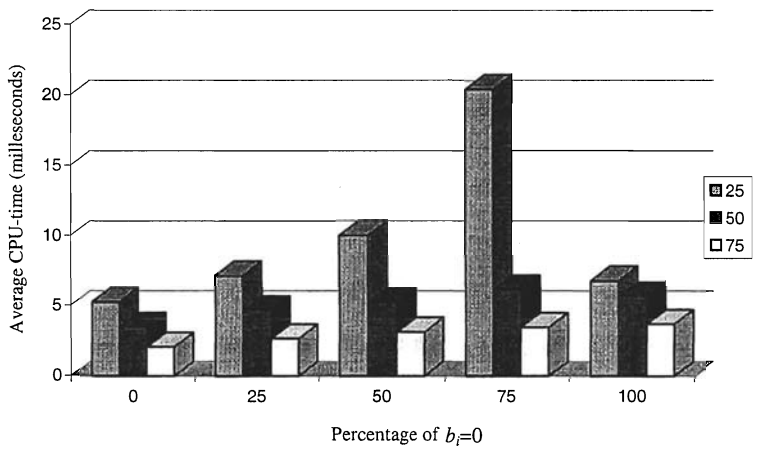

Fig. 15. The impact of the percentage of a_i<0 and b_i=0 on the average CPU-time

Fig. 16 displays the average CPU-time for different settings of the deadline δ_n of the project. The higher the deadline of the project, the higher the probability that the net present value function of each activity is divided in three regions as shown in Fig. 3. On the contrary, with a rather small deadline it is likely that the point where the net present value is minimized, i.e. $\frac{-b_i - a_i \ln \beta}{b_i \ln \beta}$, is higher than the latest finish time. In that case, the problem becomes easier since then less sets of activities *SA* will be added to the set of delaying trees *DT*.

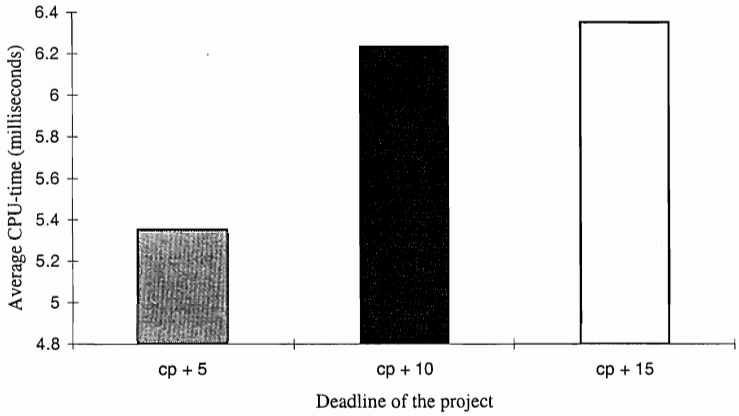

Fig. 16. The impact of the deadline on the average CPU-time

Finally, Table IV displays the average number of nodes in the search tree. Even instances with 120 activities can be solved with an average of 5 nodes in the search tree, resulting in the very low CPU-times of Table III.

Table IV. The average number of nodes in the search tree

# activities	# problems	Average number of nodes
30	21,600	1.494
60	21,600	2.540
90	21,600	4.605
120	27,000	5.120

7. Conclusions

This paper reports on an exact solution procedure for problem $cpm,\delta_n,c_j^{lin} \mid npv$, i.e. the unconstrained project scheduling problem with discounted cash flows where the net cash flows are linear dependent on their realization time. In this activity-on-the-node problem, each activity has a known deterministic net cash flow which is assumed to be linear dependent and non-increasing in time. In order to maximize the net present value of the project, the activities have to be scheduled subject to their precedence constraints and a fixed deadline which may not be exceeded.

The new exact procedure consists of an enumerative algorithm in which a recursive search procedure is embedded. The recursive search repetitively identifies sets of activities for which a forward shift increases the net present value of the project. Due to the time-dependency of the net cash flows of the project, an enumeration is needed in order to identify possible shifts which are not detected by the recursive search procedure but lead nevertheless to an increase of the net present value. Through the use of dominance rules, the number of nodes in the enumeration tree decreases considerably.

The procedure has been coded in Visual C++, version 4.0 under Windows NT 4.0 and has been tested on the well-known data set from the project scheduling library PSPLIB

(Kolisch and Sprecher, 1996). The results of extensive computational tests obtained on a Dell personal computer (Pentium III 550 MHz processor) reveal that the new exact algorithm is very efficient. This holds the promise that the procedure may be used or can be extended to other kinds of time-dependent cash flow patterns, which will be a topic of our future research.

References

Dayanand, N. and Padman, R., 1993a, "The payment scheduling problem in project networks", Working Paper 9331, The Heinz School, CMU, Pittsburgh, PA 15213

Dayanand, N. and Padman, R., 1993b, "Payments in projects: a constractor's model", Working Paper 9371, The Heinz School, CMU, Pittsburgh, PA 15213

Dayanand, N. and Padman, R., 1997, "On modeling payments in project networks", *Journal of the Operational Research Society*, 48, 906-918.

Demeulemeester, E., Herroelen, W. and Van Dommelen, P., 1996, "An optimal recursive search procedure for the deterministic unconstrained max-*npv* project scheduling problem", Research Report 9603, Department of Applied Economics, Katholieke Universiteit Leuven.

De Reyck, B. and Herroelen, W., 1996, "On the use of the complexity index as a measure of complexiy in activity networks", *European Journal of Operational Research*, 91, 347-366.

Elmaghraby, S.E. and Herroelen, W., 1990, "The scheduling of activities to maximize the net present value of projects", *European Journal of Operational Research*, 49, 35-49.

Grinold, R.C., 1972, "The payment scheduling problem", *Naval Research Logistics Quarterly*, 19, 123-136.

Herroelen, W., Demeulemeester, E. and De Reyck, B., 1998, "A classification scheme for project scheduling problems", in: Weglarz J. (Ed.), *Handbook on Recent Advances in Project Scheduling*, Kluwer Academic Publishers, Chapter 1, 1-26.

Herroelen, W., Demeulemeester, E. and Van Dommelen, P., 1997, "Project network models with discounted cash flows: A guided tour through recent developments", *European Journal of Operational Research*, 100, 97-121.

Herroelen, W. and Gallens, E., 1993, "Computational experience with an optimal procedure for the scheduling of activities to maximize the net present value of projects", *European Journal of Operational Research*, 65, 274-277.

Etgar, R. and Shtub, A., 1999, "Scheduling project activities to maximize the net present value - the case of linear time dependent, contingent cash flows", *International Journal of Production Research*, 37, 329-339.

Etgar, R., Shtub, A. and LeBlanc, L.J., 1996, "Scheduling projects to maximize net present value - the case of time-dependent, contingent cash flows", *European Journal of Operational Research*, 96, 90-96.

Kazaz, B. and Sepil, C., 1996, "Project scheduling with discounted cash flows and progress payments", *Journal of the Operational Research Society*, 47, 1262-1272.

Kolisch, R., Sprecher, A. and Drexl, A., 1995, "Characterization and generation of a general class of resource-constrained project scheduling problems", *Management Science*, 41, 1693-1703.

Kolisch, R. and Sprecher, A., 1996, "PSPLIB - A project scheduling library", *European Journal of Operational Research*, 96, 205-216.

Russell, A.H., 1970, "Cash flows in networks", *Management Science*, 16, 357-373.

Sepil, C. and Ortaç, N., 1997, "Performance of the heuristic procedures for constrained projects with progress payments", *Journal of the Operational Research Society*, 48, 1123-1130.

Shtub, A. and Etgar, R., 1997, "A branch-and-bound algorithm for scheduling projects to maximize net present value: the case of time dependent, contingent cash flows", *International Journal of Production Research*, 35, 3367-3378.

Vanhoucke, M., Demeulemeester, E. and Herroelen, W., 1999, "On maximizing the net present value of a project under resource constraints", Research Report 9915, Department of Applied Economics, Katholieke Universiteit Leuven.

APPENDIX

In this appendix, we give a detailed description of the recursive algorithm of the example from the paper. More precisely, we explore step 3 of the recursive algorithm (denoted in the paper by recursive_search procedure) of node 0 displayed in Fig. 12. In Fig. 17, we show the current tree at the end of step 2 of the recursive algorithm. We now continue with step 3, the generation of the set of delaying trees.

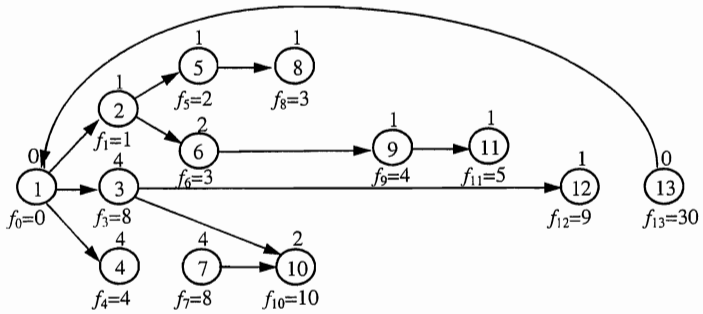

Fig. 17. The current tree *CT* as displayed in Fig. 10

STEP 3.
RECURSION(1)
SA={1}, CA={1}
RECURSION(4)
SA={4}, CA={1,4}
SA'={4}
a'=-5.00, b'=-1.00, $v_{4,7}$=0, w=20
$G(f_4)<G(f_4+w)$, $CT=CT\backslash\{1,4\}$
Update DT={(4)}

The set SA'={4} is added to the set *DT*. We update the current tree as shown in Fig. 18 and repeat the recursive_search procedure.

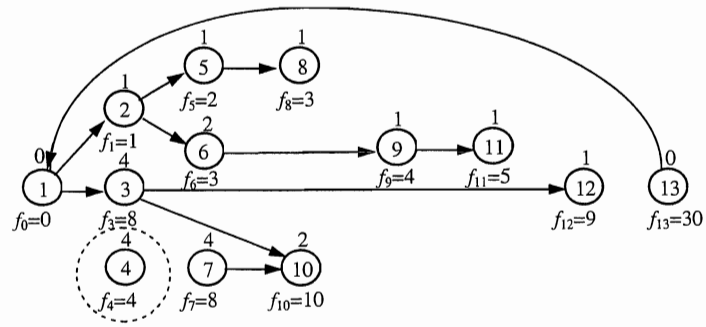

Fig. 18. The updated current tree *CT* and the set DT={(4)}

STEP 3.

RECURSION(1)
SA={1}, CA={1}
RECURSION(3)
SA={3}, CA={1,3}
RECURSION(12)
SA={12}, CA={1,3,12}
SA'={3,12}
a'=8.60, b'=0.00, $v_{12,13}$=21, w=21
RECURSION(10)
SA={10}, CA={1,3,10,12}
RECURSION(7)
SA={7}, CA={1,3,7,10,12}
SA'={7,10}
a'=0.38, b'=0.00
SA'={3,7,10,12}
a'=8.91, b'=0.00, $v_{10,13}$=20, w=20
SA'={1,3,7,10,12}
a'=3.84, b'=0.00, $v_{10,13}$=20, w=21
RECURSION(2)
SA={2}, CA={1,2,3,7,10,12}
RECURSION(6)
SA={6}, CA={1,2,3,6,7,10,12}
RECURSION(9)
SA={9}, CA={1,2,3,6,7,9,10,12}
RECURSION(11)
SA={11}, CA={1,2,3,6,7,9,10,11,12}
SA'={11}
a'=-1.00, b'=-0.25, $v_{11,12}$=3, w=24
$G(f_{11})<G(f_{11}+v_{11,12})$, $CT=CT\backslash\{9,11\}$
Update $CT=CT\cup\{11,12\}$ and f_{11}=8

Acitivity 11 is shifted forward in time. We update the current tree as shown in Fig. 19 and repeat the recursive_search procedure.

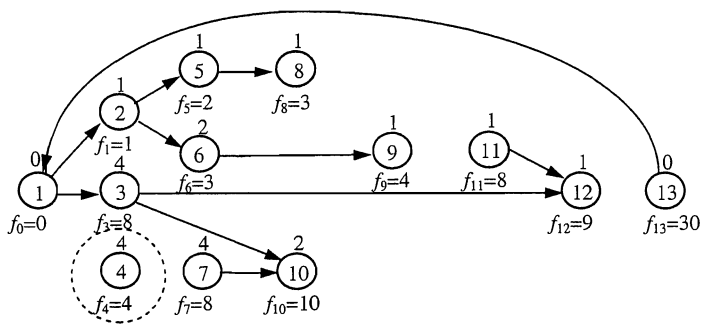

Fig. 19. The updated current tree CT with activity 11 shifted forward in time

STEP 3.

RECURSION(1)

SA={1}, CA={1}

RECURSION(3)

SA={3}, CA={1,3}

RECURSION(12)

SA={12}, CA={1,3,12}

RECURSION(11)

SA={11}, CA={1,3,11,12}

SA'={11,12}

a'=3.17, b'=-0.28

SA'={3,11,12}

a'=7.60, b'=-0.25, $v_{12,13}$=21, w=21

RECURSION(10)

SA={10}, CA={1,3,10,11,12}

RECURSION(7)

SA={7}, CA={1,3,7,10,11,12}

SA'={7,10}

a'=0.38, b'=0.00

SA'={3,7,10,11,12}

a'=7.91, b'=-0.25, $v_{10,13}$=20, w=20

SA'={1,3,7,10,11,12}

a'=2.54, b'=-0.11, $v_{10,13}$=20, w=21

RECURSION(2)

SA={2}, CA={1,2,3,7,10,11,12}

RECURSION(6)

SA={6}, CA={1,2,3,6,7,10,11,12}

RECURSION(9)

SA={9}, CA={1,2,3,6,7,9,10,11,12}

SA'={6,9}

a'=0.40, b'=0.00, $v_{9,11}$=3, w=24

SA'={2,6,9}

a'=-2.68, b'=-0.50, $v_{9,11}$=3, w=24

RECURSION(5)

SA={5}, CA={1,2,3,5,6,7,9,10,11,12}

RECURSION(8)

SA={8}, CA={1,2,3,5,6,7,8,9,10,11,12}

SA'={5,8}

a'=-4.10, b'=-1.00, $v_{8,9}$=0, w=24

$G(f_5)<G(f_5+w)$, $CT=CT\backslash\{2,5\}$

Update DT={(4),(5,8)}

The set SA'={5,8} is added to the set DT. We update the current tree as shown in Fig. 20 and repeat the recursive_search procedure.

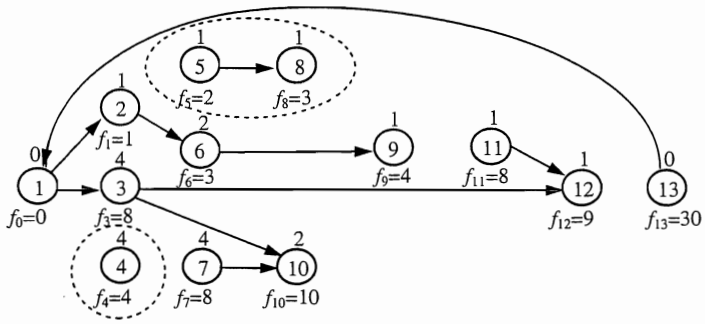

Fig. 20. The updated current tree CT and the set DT={(4),(5,8)}

STEP 3.

RECURSION(1)

SA={1}, CA={1}

RECURSION(3)

SA={3}, CA={1,3}

RECURSION(12)

SA={12}, CA={1,3,12}

RECURSION(11)

SA={11}, CA={1,3,11,12}

SA'={11,12}

a'=3.17, b'=-0.28

SA'={3,11,12}

a'=7.60, b'=-0.25, $v_{12,13}$=21, w=21

RECURSION(10)

SA={10}, CA={1,3,10,11,12}

RECURSION(7)

SA={7}, CA={1,3,7,10,11,12}

SA'={7,10}

a'=0.38, b'=0.00

SA'={3,7,10,11,12}

a'=7.91, b'=-0.25, $v_{10,13}$=20, w=20

SA'={1,3,7,10,11,12}

a'=2.54, b'=-0.11, $v_{10,13}$=20, w=21

RECURSION(2)

SA={2}, CA={1,2,3,7,10,11,12}

RECURSION(6)

SA={6}, CA={1,2,3,6,7,10,11,12}

RECURSION(9)

SA={9}, CA={1,2,3,6,7,9,10,11,12}

SA'={6,9}

a'=0.40, b'=0.00, $v_{9,11}$=3, w=24

SA'={2,6,9}

a'=2.68, b'=-0.50, $v_{9,11}$=3, w=24

$G(f_2)<G(f_2+w)$, $CT=CT\backslash\{1,2\}$

Update DT={(4),(5,8),(2,6,9)}

The set $SA'=\{2,6,9\}$ is added to the set DT. We update the current tree as shown in Fig. 21 and repeat the recursive_search procedure.

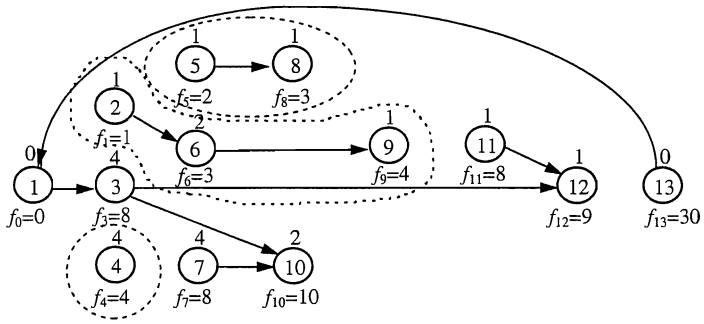

Fig. 21. The current tree CT and the set DT as displayed in Fig. 11

STEP 3.
RECURSION(1)
$SA=\{1\}$, $CA=\{1\}$
RECURSION(3)
$SA=\{3\}$, $CA=\{1,3\}$
RECURSION(12)
$SA=\{12\}$, $CA=\{1,3,12\}$
RECURSION(11)
$SA=\{11\}$, $CA=\{1,3,11,12\}$
$SA'=\{11,12\}$
a'=3.17, b'=-0.28
$SA'=\{3,11,12\}$
a'=7.60, b'=-0.25, $v_{12,13}$=21, w=21
RECURSION(10)
$SA=\{10\}$, $CA=\{1,3,10,11,12\}$
RECURSION(7)
$SA=\{7\}$, $CA=\{1,3,7,10,11,12\}$
$SA'=\{7,10\}$
a'=0.38, b'=0.00
$SA'=\{3,7,10,11,12\}$
a'=7.91, b'=-0.25, $v_{10,13}$=20, w=20
$SA'=\{1,3,7,10,11,12\}$
a'=2.54, b'=-0.11, $v_{10,13}$=20, w=21
RECURSION(13)
$SA=\{13\}$, $CA=\{1,3,7,10,11,12,13\}$
$SA'=\{1,3,7,10,11,12,13\}$
a'=2.54, b'=-0.11

No sets of activities SA can be found to shift forward in time or add to the set of delaying trees DT. The resursive_search procedure termintes. Since $DT\neq\varnothing$ the binary search will be started, as explained in the paper.